3.4 Additional Uses of Validation Invocations and Exploratory Expression Evaluation

An important part of refining a specification is translating user-level requirements, stated in English prose, into Boolean logic. Exploratory expression evaluation, including validation invocations, can be useful in this translation process.

The following are typical user-level requirements for an operation like adding a record to a database, i.e., the `AddUser` operation described in the previous section of the thesis:

- There is no user record in the input database with the same id as the record to be added; this is a a *no duplicates requirement*.
- The id of an added user record cannot be empty and must be 8 characters or less fewer in length; this is an *id syntax constraint*.
- If the area code and phone number are present, they must be 3 digits and 7 digits respectively; these are *phone number format constraints*.

Figure 3.29 contains a sample specification of a flawed AddUser precondition, The intent of the precondition logic is to define these requirements. This sample characterizes the kind of logic oversights that have been observed regularly in students' initial efforts to translate user-level requirements from English prose into formal logic.

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    precondition:
        (*
         * There is no user record in the input UserDB with the same id as the
         * record to be added.
         *)
       (not (ur in udb))

            and

        (*
         * The id of the given user record is not empty and 8 characters or
         * less.
         *)
        (#(ur.id) <= 8)

            and

        (*
         * If the phone area code and number are present, they must be 3 digits
         * and 7 digits respectively.
         *)
        (#(ur.phone.area) = 3) and
        (#(ur.phone.num) = 7);


    postcondition: (* Same as above *);

end AddUser;
```

Figure 3.29: Flawed attempt at AddUser precondition.

Figure 3.30 has corrected logic, for comparison purposes.

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    precondition:
        (*
         * There is no user record in the input UserDB with the same id as the
         * record to be added.
         *)
        (not (exists (ur' in udb) ur'.id = ur.id))

            and

        (*
         * The id of the given user record is not empty and 8 characters or
         * less.
         *)
        (ur.id != nil) and (#(ur.id) <= 8)

            and

        (*
         * If the phone area code and number are present, they must be 3 digits
         * and 7 digits respectively.
         *)
        (if (ur.phone.area != nil) then (#(ur.phone.area) = 3)) and
        (if (ur.phone.num != nil) then (#(ur.phone.num) = 7));


    postcondition: (* Same as above *);

end AddUser;
```

Figure 3.30: Improved AddUser precondition.

As with any form of debugging, there are a variety of ways to test and correct flaws in logic. Validation invocations provide a useful tool that can help in the process. In the example at hand, each flaw can be revealed with a single, reasonably straightforward validation invocation.

The first flaw is the translation of the English requirement *"There is no user record in the input UserDB with the same id as the record to be added."* The flawed versus correct versions of the logic are

```
(not (ur in udb))
```

versus

```
(not (exists (ur' in udb) ur'.id = ur.id))
```

This flaw can be detected with a validation condition that attempts to add a user record with the same id, but different name, to the database. E.g.,

```
val phone:PhoneNumber = {805, 5551212};
val email:EmailAddress = "pccorwin@calpoly.edu";
val ur:UserRecord = {"Corwin", "1", email, phone};
val ur_duplicate_id:UserRecord = {"Fisher", "1", email, phone};
val udb:UserDB = [];
val udb_added:UserDB = [ur];

> AddUser(udb_added, ur_duplicate_id) ?-> (udb_added);
```

The correct output of this validation is `{false,nil}`, since the precondition should fail when trying to add a record with the same id value to a database containing a record with that id, i.e., "1". The flawed logic is not strong enough, since it does not check specifically for the id value of each extant record. This kind of error is typical with students who may be initially averse to using quantifiers, and will do their best to avoid their use. A validation counter-example can succinctly illustrate the problem with the flawed logic.

The second flaw is the translation of *"The id of the given user record is not empty and 8 characters or less."* The flawed versus correct versions of the logic are:

```
(#(ur.id) <= 8)
```

versus

```
(ur.id != nil) and (#(ur.id) <= 8)
```

The problem here is that the length operator returns 0 for a nil string value. The following validation condition reveals the problem:

```
val ur_empty_id:UserRecord = {"Corwin", nil, email, phone};

> AddUser(udb, ur_empty_id) ?-> (udb);
```

The result of this evaluation should be `{false,nil}`, since the precondition should fail if the id is `nil`. Here `nil` is the translation of "empty" in the prose statement of the requirement. The flawed logic precondition evaluates to `{true,nil}`, since `#(ur.id)` = 0 when `ur.id` is `nil`, and hence `0 <= 8` evaluates to true.

To some extent, this problem has to do with the specific semantics of FMSL. However, all formal specification languages have specific rules, and users of the languages must understand clearly what the rules are. Using validation invocations and additional exploratory evaluation can help a user develop such understanding.

Some additional exploration of this example could take the following form:

```
val empty_integer:integer = nil;
val empty_string:string = nil;
obj StringList = string*;
val empty_list:StringList = nil;

> #empty_integer;
> #empty_string;
> #empty_list;
```

where all three expressions evaluate to 0. In the case of the integer value, the length operator is overloaded to evaluate to the number of integer digits. The rules illustrated here could be read in the FMSL users manual. However, the ability to explore interactively can be enlightening, as it is in the environments of interpretive and conversational programming languages.

The third and final flaw in Figure 3.29 is the translation of *"If the phone area code and number are present, they must be 3 digits and 7 digits respectively."* The flawed and correct versions of the of the logic are:

```
(#(ur.phone.area) = 3) and
(#(ur.phone.num) = 7));
```

versus

```
(if (ur.phone.area != nil) then (#(ur.phone.area) = 3)) and
(if (ur.phone.num != nil) then (#(ur.phone.num) = 7));
```

The problem is revealed with the following validation invocation:

```
val ur_empty_phone:UserRecord = {"Corwin", "1", email, nil};

> AddUser(udb, ur_empty_phone)?->(udb);
```

The correct validation result is `{true,nil}`, since the requirement allows the phone number components to be empty. Without the explicit check for this, the sub-expression `ur.phone.area` evaluates to `nil`. As explained in the previous example, the length operator applied to a `nil` value uniformly returns 0. This means that `#(ur.phone.area)` returns 0, which leads the precondition to evaluate to `false` instead of `true`.